Model-Based Testing with Graph Grammars

Vincent de Bruijn

March 8, 2012

Abstract

Graph Grammars have many structural advantages, which are potential benefits for the model-based testing process.This report describes the setup of a research project where the use of Graph Grammars in model-based testing is researched. The goal of the project is to create a system for automatic test generation from Graph Grammars and to validate this system. A graph transformation tool, GROOVE, and a model-based testing tool, ATM, are used as the backbone of the system. The system will be validated using the results of several case-studies.

Contents

1 Introduction

In software development projects, often limited time and resources are available for testing. However, testing is an important part of software development, because it decreases future maintainance costs [13]. Testing is a complex process and should be done often [16]. Therefore, the testing process should be as efficient as possible in order to save resources.

Test automation allows repeated testing during the development process. The advantage of this is that bugs are found early and can therefore be fixed early. A widely used practice is maintaining a *test suite*, which is a collection of tests. However, when the creation of a test suite is done manually, this still leaves room for human error [10]. Also, manual creation of test-cases is not time-efficient.

Creating an abstract representation or a *model* of the system is a way to tackle these problems. What is meant by a model in this report, is the description of the behavior of a system. Models such as state charts and sequence charts, which only describe the system architecture, are not considered here. A model can be used to systematically generate tests for the system. This is referred to as *model-based testing*. This leads to a larger test suite in a shorter amount of time than if done manually. These models are created from the specification documents provided by the end-user. These specification documents are 'notoriously error-prone' [12]. If the tester copies an error in the document or makes a wrong interpretation, the constructed model becomes incorrect.

The stakeholders evaluate the constructed model to verify its correctness. However, the visual or textual representation of large models may become troublesome to understand, which is referred to as the model having a low model transparency. The feedback process of the stakeholders is obstructed by low transparency models. Models that are often used are state machines, i.e. a collection of nodes representing the states of the system connected by transitions representing an action taken by the system. The problem in such models with a large amount of states is the decrease of model transparency. Errors in models with a low transparency are not easily detected.

A formalism that among other things can describe software systems is Graph Transformation. The system states are represented by graphs and the transitions between the states are accomplished by applying graph change rules to those graphs. These rules can be expressed as graphs themselves. A graph transformation model of a software system is therefore a collection of graphs, each a visual representation of one aspect of the system. This formalism may therefore provide a more intuitive approach to system modelling. Graph Transformation and its potential benefits have been studied since the early '70s. The usage of this computational paradigm is best described by the following quote from Andries et al. [1]: "Graphs are well-known, well-understood, and frequently used means to represent system states, complex objects, diagrams, and networks, like flowcharts, entity-relationship diagrams, Petri nets, and many more. Rules have proved to be extremely useful for describing computations by local transformations: Arithmetic, syntactic, and deduction rules are well-known examples." An informative paper on graph transformations is written by Heckel et al. [7]. A quote from this paper: "Graphs and diagrams provide a simple and powerful approach to a variety of problems that are typical to computer science in general, and software engineering in particular."

1.1 Research setup

This report describes the setup of a project where the use of Graph Grammars in model-based testing is researched. In the introduction the motivation is given for using this type of modelling technique. The goal is to create a system for automatic test generation on Graph Grammars. If the assumptions that Graph Grammars provide a more intuitive modelling and testing process hold, this new testing approach will lead to a more efficient testing process and fewer incorrect

models. The system design, once implemented and validated, provides a valuable contribution to the testing paradigm.

Tools that perform statespace exploration on Graph Grammars and tools for automatic test generation already exist. Two of these tools will be used to perform these functions. The graph transformation tool GROOVE[1] will be used to model and explore the Graph Grammar. The testing tool developed by Axini[2] is used for the automatic test generation. This tool has no name, but will be referred to in this report as Axini Test Manager (ATM).

1.2 Research questions

The research questions are split into a design and validation component:

1. **Design**: How can automatic test generation be done using graph transformation systems? In particular, how can ATM be used together with GROOVE to achieve this?
2. **Validation**: What are the strengths and weaknesses of using graph transformation systems in model-based testing?

The result of the design question will be one system which incorporates ATM and GROOVE. This system will be referred to as the GROOVE-Axini Testing System (GRATiS). In order to answer the first research question sufficiently, GRATiS should produce tests on the basis of a Graph Grammar and give correct verdicts whether the system contains errors or not.

The criteria used to assess the strengths and weaknesses are split into two parts: the objective and the subjective criteria. The objective criteria are the measurements that can be done on GRATiS, such as speed and statespace size. The subjective criteria are related to how easy graph transformation models are to use and maintain. The assessment of the latter criteria requires a human component. The criteria and methods for the assessment are elaborated in section 6.2.

GRATiS will be validated using case-studies. These case-studies are done with existing specifications from systems tested by Axini. Each case-study will have a Graph Grammar and a symbolic model which describe the same system. GRATiS and ATM are used for the automatic test generation on these models respectively. Both the model and the test process will then be compared as part of the validation. This includes the criteria mentioned in this section. This is explained in more detail in section 6.2.

1.3 Structure of the report

The structure of this document is as follows: the general model-based testing process is set out in section 2. Section 3 describes symbolic models used in ATM and section 4 describes Graph Grammars used in GROOVE. GROOVE and ATM are introduced in section 5. The method used to answer the research questions above is in section 6. The planning for the project is in appendix ??.

2 Model-based Testing

Formal testing theory was introduced by De Nicola et al. [15]. This testing theory was first used in algorithms for automatic test generation by Brinksma [3]. Tretmans gives a formal approach to protocol conformance testing (whether a protocol conforms to its specifications) in [19]. Model-based testing is the process of automatically generating test-cases based on a specification. A

[1] http://sourceforge.net/projects/groove/
[2] http://www.axini.nl/

good overview of model-based testing theory and past research is given in "Model-Based Testing of Reactive Systems" [11].

Model-based testing is a testing technique where a System Under Test (SUT) is tested for conformance to a model description of the system. The general setup for this process is depicted in Figure 1. The specification of a system, given as a model, is given to a test derivation component which generates test cases. These test cases are passed to a component that executes the test cases on the SUT. Tests are executed by providing input/stimuli to the SUT and monitoring the output/response. The test execution component evaluates the test cases, the stimuli and the responses. It gives a 'pass' or 'fail' verdict whether the SUT conforms to the specification or not respectively.

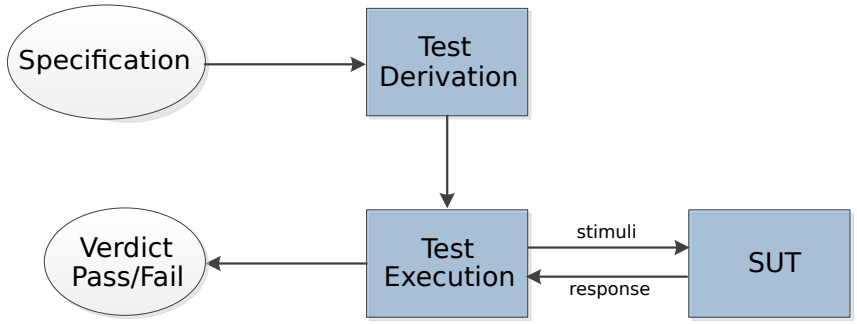


Figure 1: A general model-based testing setup

This type of model-based testing is called *batch testing* or *offline testing*. Another type of model-based testing is *on the fly* testing. The main difference is that no test cases are derived, instead a transition in the model is chosen and tested on the system directly. The general architecture for this process is shown in Figure 2. A tool for on-the-fly testing is TorX [18], which integrates automatic test generation, test execution, and test analysis. A version of this tool written in Java under continuous development is JTorX [2].

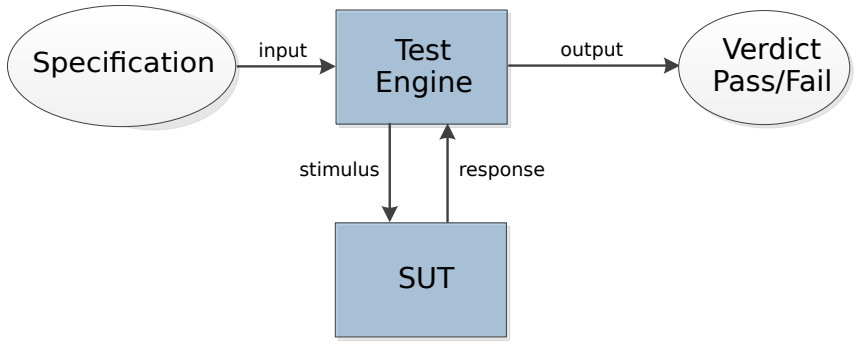


Figure 2: A general 'on-the-fly' model-based testing setup

Variations of state machines and transition systems have been widely used as the underlying model for test generation. Other tools use the structure of data types to generate test data. First, two types of models are introduced. These are basic formalisms useful to understand the models in the rest of the paper. Then, the notion of *coverage* is explained.

2.1 Labelled Transition Systems

A labelled transition system is a structure consisting of states with labelled transitions between them.

Definition 2.1.1. A labelled transition system is a 4-tuple $\langle Q, L, T, q0 \rangle$, where:

- Q is a finite, non-empty set of states
- L is a finite set of labels
- $T \in Q \times (L \cup \{\tau\}) \times Q$, with $\tau \notin L$, is the transition relation
- $q0 \in Q$ is the initial state.

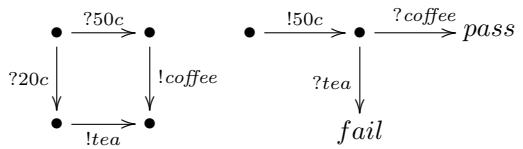
We write $q \xrightarrow{\mu} q'$ if there is a transition labelled μ from state q to state q', i.e., $(q, , q') \in T$. The informal idea of such a transition is that when the system is in state q it may perform action μ, and go to state q'.

2.2 Input-Output Transition Systems

A useful transition system for model-based testing is the Input-Output Transition System (IOTS) by Tretmans[20]. Assuming that implementations communicate with their environment via inputs and outputs, this formalism is useful for describing system behavior. These are similar to LTSs with the exception that labels have a type. This type can be 'input' or 'output'.

An example of such an IOTS is shown in Figure 3a. This system allows an input of 20 or 50 cents and then outputs tea or coffee accordingly. The inputs are preceded by a question mark, the outputs are preceded by an exclamation mark. This system is a specification of a coffee machine. A test case can also be described by an IOTS. A test case for the coffee machine is given in Figure 3b. The test case is derived from the coffee machine IOTS in the Test Derivation component. The Test Execution component applies the stimulus '50c' to the SUT. When the SUT responds with 'tea', the test case results in a fail verdict and when the SUT responds with 'coffee', the test case results in a pass verdict. The pass and fail verdicts are two special states in the test case, which are sink states, i.e., once in either of those the test case cannot leave that state.

Test cases should always reach a pass or fail state within finite time. This requirement ensures that the testing process halts.



(a) An IOTS (b) An IOTS test case

Figure 3: The specification of a coffee machine and a test case as an IOTS

2.3 Coverage

The amount of tests that can be generated from a model is potentially infinite. Therefore, there must be a test selection strategy to maximize the quality of the tests while minimizing the time spent testing. Coverage statistics help with test selection. This statistic indicates how much of the SUT is tested. When the SUT is a black-box, typical coverage metrics are state and transition coverage of the model [9, 14, 6].

As an example, the coverage metrics of the IOTS test case example in 3b are calculated. The test case tests one path through the specification and passes through 3 out of 4 states and 2 out of 4 transitions. The state coverage is therefore 75% and the transition coverage is 50%.

3 Symbolic Transition Systems

Symbolic test generation is introduced by Rusu et al. [17], using Input-Output Symbolic Transition Systems (IOSTSs). Symbolic Transition Systems (STSs) are introduced by Frantzen et al. [8]. A tool that generates tests based on symbolic specification is the STG tool, described in Clarke et al. [4].

In this section, STSs are introduced which combine LTSs with a data type oriented approach. Among other tools, they are used in practice in ATM.

An STS has *locations* and *switch relations*. If the STS represents a model of a software system, a location in the STS represents a state of the system, not including data values. A switch relation defines the transition from one location to another. The *location variables* are a representation of the data values in the system. A switch relation has a *gate*, which is a label representating the execution steps of the system. Gates also have *interaction variables*, which are data values specific to the gate. Switch relations also have *guards* and *update mappings*. A guard is a boolean function, which disallows using the switch relation when the function evaluates to false. An update mapping is a mapping of location variables to an arithmetic function. After the system switches to a new location, the variables in the update mapping will have the value corresponding to the evaluation of the arithmetic function they map to. We define $\mathcal{F}(\mathcal{T})$ to be a collection of boolean functions over a set of terms $\mathcal{T}$. These terms are variables and basic data types, such as integers, strings and arrays.

We define $\mathcal{G}(\mathcal{T})$ to be a collection of arithmetic functions over a set of terms $\mathcal{T}$.

3.1 STS Definition

The definition of an STS that follows is based on the definition given by Frantzen et al. [8]. A Symbolic Transition System is a tuple $\langle L, l_0, \mathcal{V}, \imath, \mathcal{I}, \Lambda, \rightarrow \rangle$, where:

- L is a finite set of locations and $l_0 \in L$ is the initial location.
- $\mathcal{V}$ is a finite set of location variables.
- $\imath$ is a mapping $\{v \mapsto x \mid v \in \mathcal{V}, x \in \mathcal{X}\}$ where $\mathcal{X}$ is a collection of basic data types, representing the initialisation of the location variables.
- $\mathcal{I}$ is a set of interaction variables, disjoint from $\mathcal{V}$.
- Λ is a finite set of gates. The unobservable gate is denoted $\tau (\tau \notin \Lambda)$; we write Λ_τ for $\Lambda \cup \{\tau\}$. The arity of a gate $\lambda \in \Lambda_\tau$, denoted $arity(\lambda)$, is a natural number. The parameters of a gate $\lambda \in \Lambda_\tau$, denoted $param(\lambda)$, are a tuple of length $arity(\lambda)$ of distinct interaction variables. We fix arity$(\tau) = 0$, i.e. the unobservable gate has no interaction variables.
- $\rightarrow \subseteq L \times \Lambda_\tau \times \mathcal{F}(\mathcal{V} \cup \mathcal{I} \cup \mathcal{X}) \times \{v \mapsto x \mid v \in \mathcal{V}, x \in \mathcal{G}(\mathcal{V} \cup \mathcal{I} \cup \mathcal{X})\} \times L$, is the switch relation. We write $l \xrightarrow{\lambda, \phi, \rho} l'$ instead of $(l, \lambda, \phi, \rho, l') \in \rightarrow$, where ϕ is referred to as the guard and ρ as the update mapping. We require $var(\phi) \cup var(\rho) \subseteq \mathcal{V} \cup param(\lambda)$, where var is the collection of the variables used in the given guard or update mapping.

An IOSTS can now easily be defined. The same difference between LTSs and IOTSs applies, namely each gate in an IOSTS has a type, 'input' or 'output'. As with IOTSs, each gate has a '?' or '!' to indicate whether it is an input or an output respectively.

3.2 IOSTS example

In Figure 4 a simple board game is shown, where two players consecutively throw a die and move along four squares. The 'init' switch relation is a graphical representation of the variable initialization $\imath$. The defining tuple of the IOSTS is:

$\langle \{throw, move\}, throw, \{T, P, D\}, \{T \mapsto 0, P \mapsto [0,2], D \mapsto 0\}, \{d, p, l\}, \{throw?, move!\},$
$\{throw \xrightarrow{throw?,1<=d<=6,D\mapsto d} move, move \xrightarrow{move!,T=p\wedge l=(P[p]+D)\%4,P[p]\mapsto l,T\mapsto p\%2} throw\}\rangle$

The variables T, P and D are the location variables symbolizing the player's turn, the positions of the players and the number of the die thrown respectively. The output gate *move*! has $param = \langle p, l\rangle$ symbolizing which player moves to which location. The input gate *throw*? has $param = \langle d\rangle$ symbolizing which number is thrown by the die. The switch relation with gate *throw*? has the restriction that the number of the die thrown is between one and six and the update sets the location variable D to the value of interaction variable d. The switch relations with gate *move*! have the restriction that it must be the turn of the player moving and that the new location of the player is the number of steps ahead as thrown by the die. The update mapping sets the location of the player to the correct value and passes the turn to the next player. In Figure 4 the gates, guards and updates are separated by pipe symbols '|' respectively.

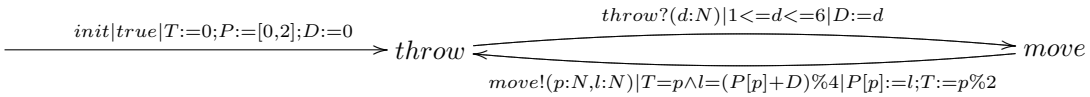


Figure 4: The STS of a board game example

3.3 STS to LTS transformation

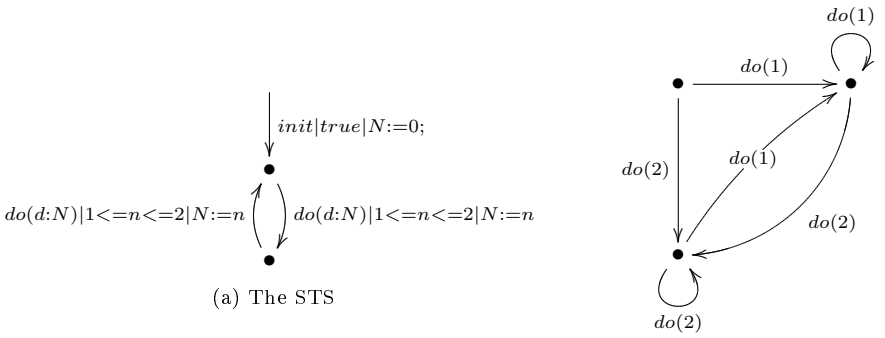
There exists a transformation of an STS to an LTS. This transformation is explained in the next paragraph and then illustrated by an example.

Consider an STS S and its transformation LTS L. A mapping between the locations, switch relations and variables of S and the states and transitions of L is given, representing the first step of the transformation. The mapping is found by:

- The unique combination of a location and a tuple of concrete values for each location variable in S maps to a state in L.
- The unique combination of a switch relation and a tuple of concrete values for each interaction variable in S maps to a transition in L.

Finding the topology of L is the next step of the transformation. For a switch relation r from location A to location B, tuples of concrete values for the location variables v_l and interaction variables v_i and a mapping of $A + v_l$ to a state s, s is the source state of a transition t, if the guard of r evaluates to true for v_l and v_i. v_u is the tuple of concrete values of the location variables after the update mapping of r is done. Then, the target state s' of t is the state mapped by $B + v_u$. The label of t is the gate of r + a textual representation of v_i. Applying this rule for the topology to all locations, switch relations and concrete values for the variables, results in L. The start state $q0$ of L is the state mapped by l_0 and $\imath$ of S. All states not reachable from $q0$ are removed from L. The range of the variables in an STS is considered to be finite, therefore the transformation is always possible to an LTS with finite number of states.

An example of this transformation is shown in Figure 5.



(a) The STS

(b) The LTS

Figure 5: An example of a transformation of an STS to an LTS

3.4 Coverage

The simplest metric to describe the coverage of an STS is the location and switch-relation coverage, which express the percentage of locations and switch relations tested in the test run. Measuring state and transition coverage of an STS is possible using the LTS resulting from the STS transformation. However, this metric is not always useful, because the number of states and transitions in the LTS depend on the number of unique combinations of concrete values of the variables in the STS. This is potentially very large. For example, when the guards of the switch relations in Figure 5a are removed, the transformation leads to an LTS with a state and transition for each possible value of an integer. It is often unfeasable to test every data value in the STS. The most interesting data values to test are determined by *boundary-value analysis* and *equivalence partitioning*. The equivalence partitions and boundaries of data values are determined by the guards of the switch relations. A method for deriving these partitions and boundaries will be part of the research project.

Moet hier nog een verwijzing bij?

Data coverage expresses the percentage of data tested in the test run, considering data to be similar if located in the same partition and a better representative of the partition if located close to the partition boundary. These properties of the tested data affect the data coverage percentage. Deriving a concrete formula for this metric will be part of the research project.

4 Graph Transformation Systems

A Graph Transformation System is composed of a start graph and a set of transition rules. The start graph describes the system in its initial state. The transition rules apply changes to the graph, creating a new graph which describes the system in its new state.

4.1 Graphs & Morphisms

A graph is a tuple $\langle L, N, E \rangle$, where:

- L is a set of labels
- N is a set of nodes, where each $n \in N$ has a label $l \in L$
- E is a set of edges, where each $e \in E$ has a label $l \in L$ and nodes $source, target \in N$

A graph H has an *occurrence* in a graph G, denoted by $L \to G$, if there is a mapping *occ* which maps the nodes and the edges of H to the nodes and the edges of G respectively.

4.2 Graph Transformation rules

A transformation rule is a tuple $\langle LHS, NAC, RHS, Map \rangle$, where:

- LHS is a graph representing the left-hand side of the rule
- NAC is a set of graphs representing the negative application conditions
- RHS is a graph representing the right-hand side of the rule
- Map is a mapping of elements in LHS to elements in RHS

A rule R is applicable on a graph G if its LHS has an occurrence in G and none of the graphs in its NAC have an occurrence in G. After the rule application, all elements in LHS not part of Map are removed from G and all elements in RHS not part of Map are added to G. All elements in Map are kept.

4.3 GTS Definition

A Graph Transformation System is a tuple $\langle G, R \rangle$, where:

- G is a graph of the start state
- R is a set of transformation rules

By applying one of the transformation rules on the graph of the start state, a new graph state is explored. These two graph states are connected by a *rule transition*, meaning the application of the rule on the start state yielding the new state. This structure resembles that of an LTS; by repeatedly applying all applicable transformation rules to each graph state until no new graph states can be explored, a transition system of graph states and rule transitions is found. This is called the *Graph Transition System* (GTiS) of the GTS. The rule transitions are labelled with a unique identifier of the rule, such as the name of the rule. This entails that each transition derived from the same rule, has the same label.

4.4 Example

The running example from Figure 4 is displayed as a GTS, as visualized in GROOVE, in Figure 6. Figure 6a is the start graph of the system. The rules can be described as follows:

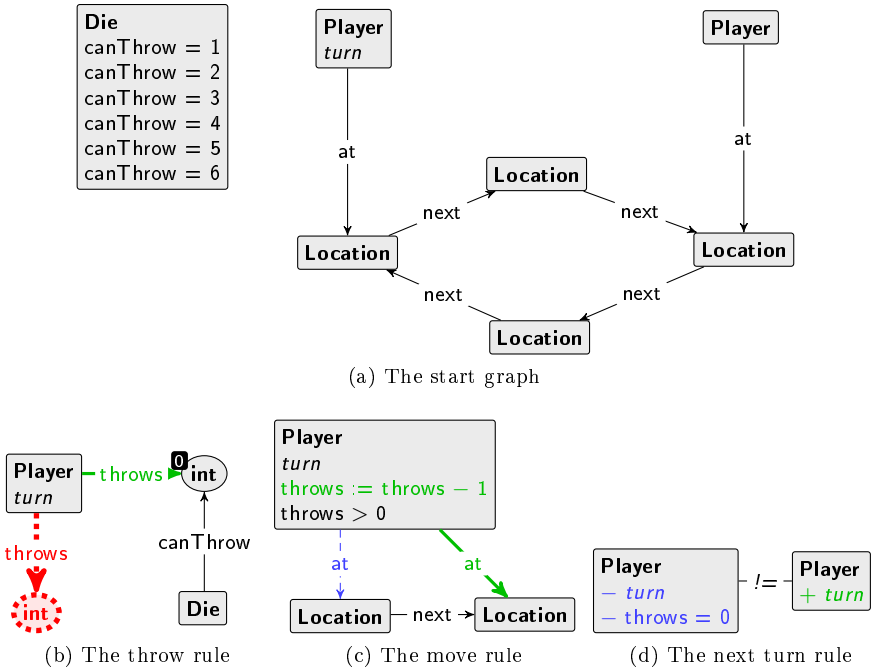
1. 6b: 'if a player has the turn and he has not thrown the die yet, he may do so.'
2. 6c: 'if a player has the turn and he has thrown the die and this number is larger than zero, he may move one place and then it is as if he has thrown one less.'
3. 6d: 'if a player has finished moving (number thrown is zero), the next player receives the turn.'

The LHS, NAC and RHS of each rule are displayed as one graph. The colored nodes and edges in the rules indicate to which part they belong:

1. normal line (black): This node or edge is part of both the LHS and RHS. Map contains the mapping of this node as part of the LHS to itself as part of the RHS.
2. dotted line (red): This node or edge is part of the NAC only.

3. thick line (green): This node or edge is part of the *RHS* only.
4. dashed line (blue): This node or edge is part of the *LHS* only.

The *turn* flag on the **Player** node is a representation of a self-edge with label *turn*. The assignments on the **Die** node are representations of edges to integer nodes. The throws value assignment (:=) in the move rule is a shorthand for two edges: one edge in the *LHS* with label *throws* from the **Player** node to an integer node with value i and another edge in the *RHS* with label *throws* from the **Player** node to an integer node with value $i - 1$. In the next turn rule, the *turn* edge exists in the *LHS* as a self-edge of the left **Player** node and in the *RHS* as a self-edge of the right **Player** node. In the same rule, the *throws* edge from the left **Player** node to an integer node only exists in the *LHS*.



(a) The start graph

(b) The throw rule

(c) The move rule

(d) The next turn rule

Figure 6: The GTS of the board game example in Figure 4

The graph is transformed after the rule is applied. The resulting graph after the transformation is the new state of the system and the rule is the transition from the old state (the graph as it was before the rule was applied) to the new state. Figure 7 shows the GTiS of one *throws* rule application on the start graph. The number on the label is the *parameter* of the label. This number is represented by the **int** node marked with a '0'. State s_1 is a representation of the graph in Figure 6a. Figure 8 shows the graph represented by s_2.

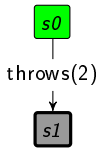


Figure 7: The GTiS after one rule application on the board game example in Figure 6

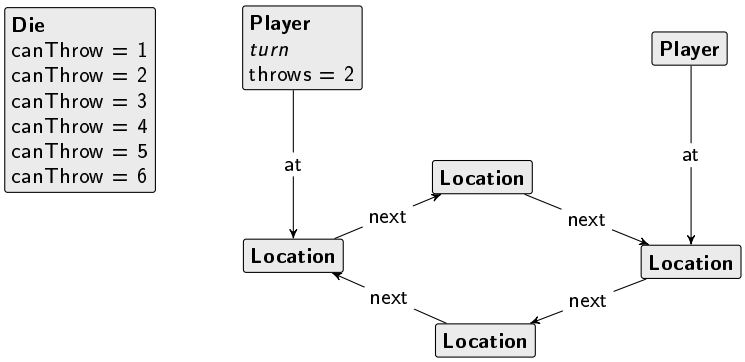


Figure 8: The graph of state $s2$ in Figure 7

5 Tooling

5.1 ATM

ATM is a web-based application, developed in the Ruby on Rails framework. It is used to test the software of several big companies in the Netherlands since 2006. It is under continuous development by Axini.

The architecture is shown graphically in Figure 9. It has a similar structure to the on-the-fly model-based testing tool architecture in Figure 2.

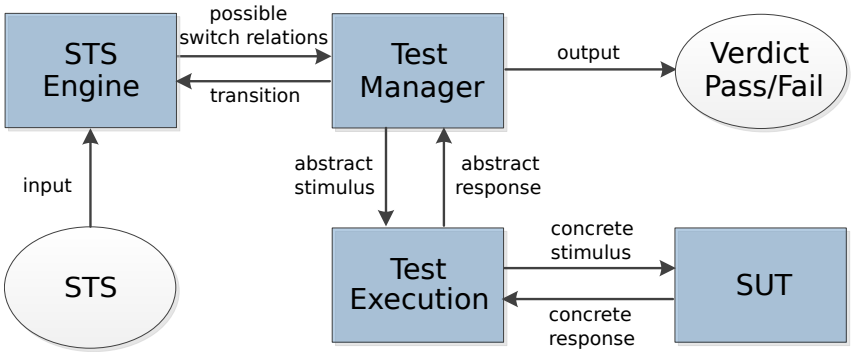


Figure 9: Architecture of ATM

The tool functions as follows:

1. An STS is given to an STS Engine, which passes the possible switch relations from the current state to the Test Manager.
2. The Test Manager chooses a switch relation and the data values, based on a test strategy. In the LTS in the STS-to-LTS transformation, this choice is represented by one transition.
3. The label of the transition is given to the Test Execution component as an *abstract stimulus*. The term abstract is used here to indicate that the transition is specific to the model. It represents some computation steps taken in the SUT. For instance, a transition with label 'connect?' is an abstract stimulus of the actual setup of a TCP connection between two distributed components of the SUT.
4. The translation of an abstract stimulus to a concrete stimulus is done by the Test Execution component. This component provides the stimulus to the SUT. When the SUT responds, the

Test Execution component translates this response to an abstract response. For instance, the Test Execution component receives an HTTP response that the TCP connect was succesful. This is a concrete response, which the Test Execution component translates to an abstract response, such as a transition with label 'ok!'. The Test Manager is notified with this abstract response.

5. The Test Manager updates the STS Engine on which transition was chosen as stimulus and which transition was detected based on the response of the SUT. If this latter transition is possible according to the model, the Test Manager gives a pass verdict for this test. Otherwise, the result is a fail verdict.

5.2 GROOVE

GROOVE is an open source, graph-based modelling tool in development at the University of Twente since 2004. It has been applied to several case studies, such as model transformations and security and leader election protocols [5].

The architecture of the GROOVE tool is shown graphically in Figure 10. A GTS is given as input to a Rule Applier component, which determines the possible rule transitions. An Exploration Strategy can be started or the user can explore the states manually using the GUI. These components request the possible rule transitions and respond with the chosen rule transition (based on the exploration strategy or the user input). The Exploration Strategy can do an exhaustive search, resulting in a GTiS. The graph states and rule transitions in this GTiS can then be inspected using the GUI.

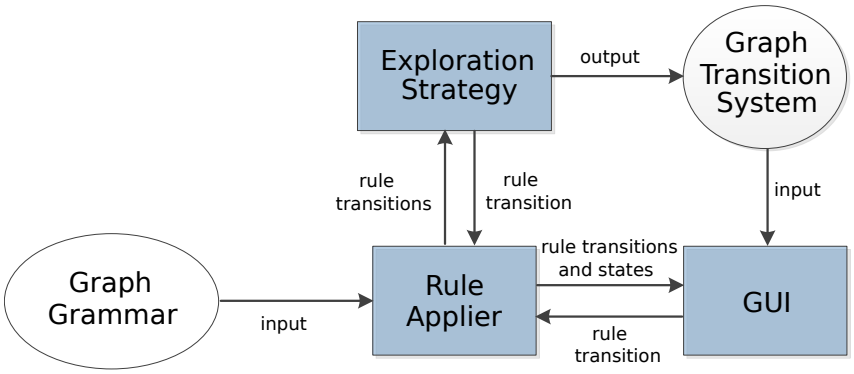


Figure 10: The GROOVE Tool

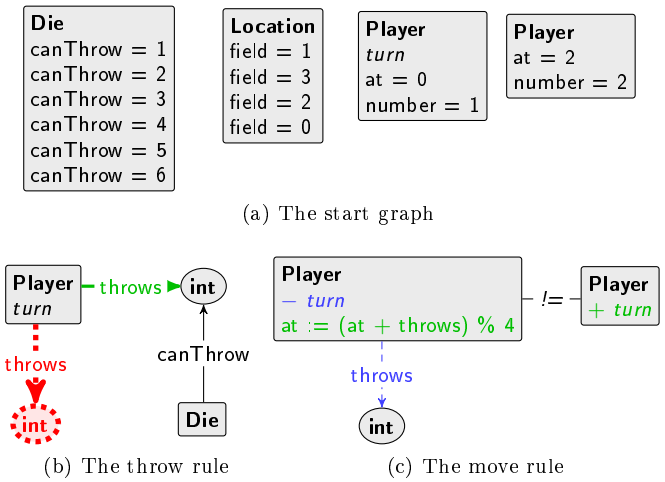
5.3 Comparison of the examples

The models of the boardgame example in Figures 4 and 6 are very different. In this section the STS and the GTS of the example are compared.

5.3.1 Comparison of behavior

The GTS of the boardgame example has a number of consecutive transitions when a player moves. The *move* rule puts the **Player** on the next **Location** and lowers the remaining 'moves' by one. This is different from the STS, which updates the location variable in one transition. The effect is that the behavior of both systems is different; one specifies the movement of a **Player** as: "Player p moves to Location l", the other as: "The Player with the turn moves to the next Location".

Which behavior is required when this boardgame would be tested depends on the implementation of the game. However, to show the power of the GTS formalism, Figure 11 shows the GTS with the same behavior as the STS. It models the location as a variable and updates this variable in one transition. It also identifies the **Players** by giving them a number.



(a) The start graph

(b) The throw rule

(c) The move rule

Figure 11: Another GTS model of the board game example

The new GTS loses many advantages by structuring it in this way: the overview of the board is gone, the rules are less visual and extending the locations in different directions is much harder. On the other hand, there are less rules and the graphs are more compact. However, when finding the GTiS corresponding to this GTS, the labels of the transitions of that GTiS do not reflect the 'move(p:N, l:N)' label of the STS. This should be done by marking the correct nodes as described in section 4.4. The problem is that the result of the equation in the 'move' rule is only derived when the rule is applied. Figure 12 shows a rule where the equation is shown graphically. The die roll, connected by the 'throws' edge, and the number of the **Location** the **Player** is at are added. The result is represented by the **int** node connected by the 'add' edge. This result modulo 4 is represented by the **int** node connected by the 'mod' edge. This node is marked as the second parameter, the number of the **Player** is marked as the first parameter. This labels the transitions with which player moves to which location.

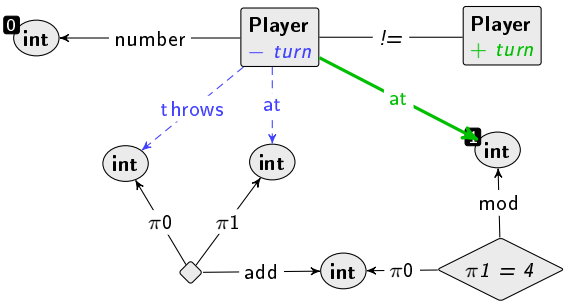


Figure 12: An alternative move rule

5.3.2 Comparison of Transition Systems

The GTiS of a GTS can be found using GROOVE and the STS can be transformed to an LTS. The two GTSs and the STS of the board game example result in three transition systems which can be compared.

The GTS from Figure 6 generates a GTiS with 32 states with 52 transitions, which can be seen visually in Figure 14. The GTS from Figure 11, using the 'move' rule from Figure 12, generates 224 states with 384 transitions, shown as GTiS in Figure 15. The reason of the difference in number of states and transitions is that the board is circular: to the first GTS, the players being at locations 1 and 3 is the same as them being at locations 2 and 4. However, this is not the same to the second GTS. Also, for the first GTS it does not matter which **Player** node is at a **Location** node; they are the same apart from which **Player** has the 'turn'. As an example, consider the start graph in Figure 6. If both players throw a '1' and move to the next location, the state is as shown in Figure 13. Both states are symmetrical and therefore they are the same state. This leads to a *symmetry reduction* of the statespace for the first GTS.

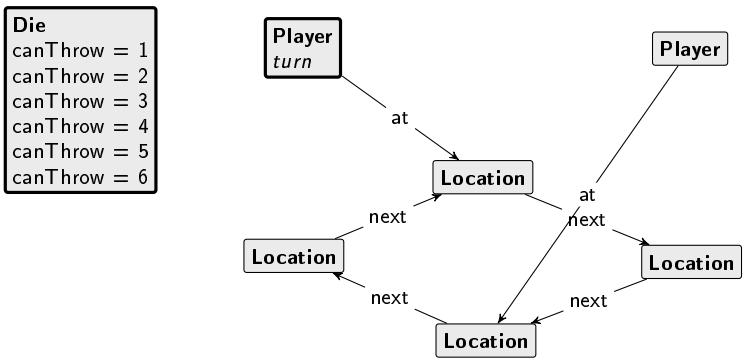


Figure 13: An example of a state symmetrical with the state in Figure 6

The LTS where the STS in Figure 4 is transformed to has 224 states and 384 transitions. This is calculated by taking all possibilities of the data values except for the die roll. This leads to 32 states ($4 \times 4 \times 2$). These 32 'throw' states each have 6 'throw?' transitions to a 'move' state, thus there are 192 'move' states. The 'move' states only have one transition back to a 'throw' state. There are $6 \times 32 + 192 \times 1 = 384$ transitions.

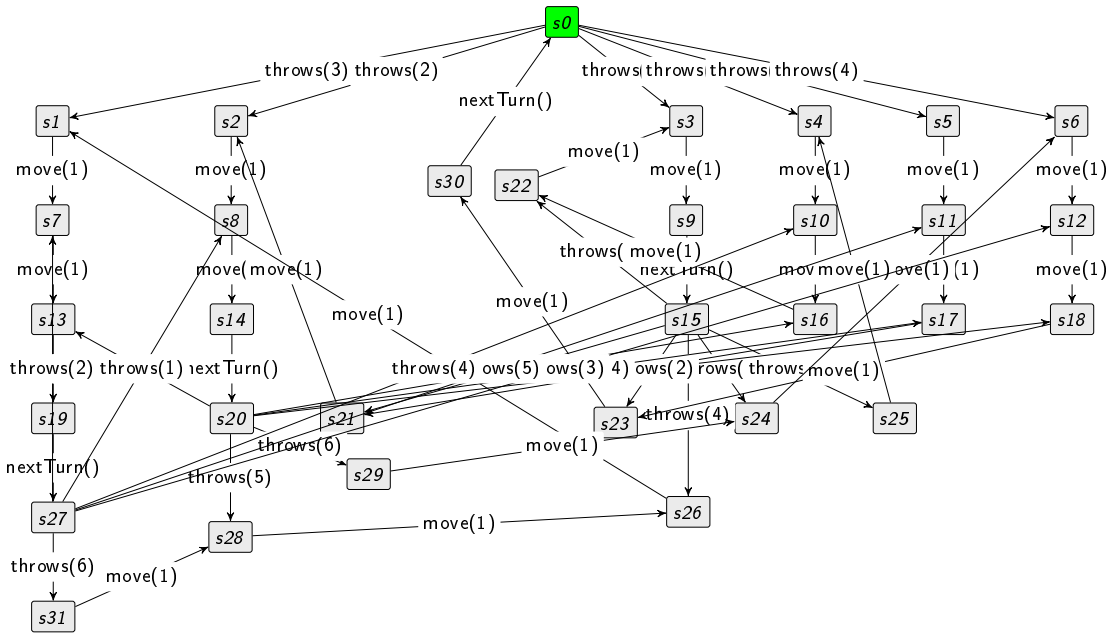


Figure 14: The GTiS of the model in Figure 6

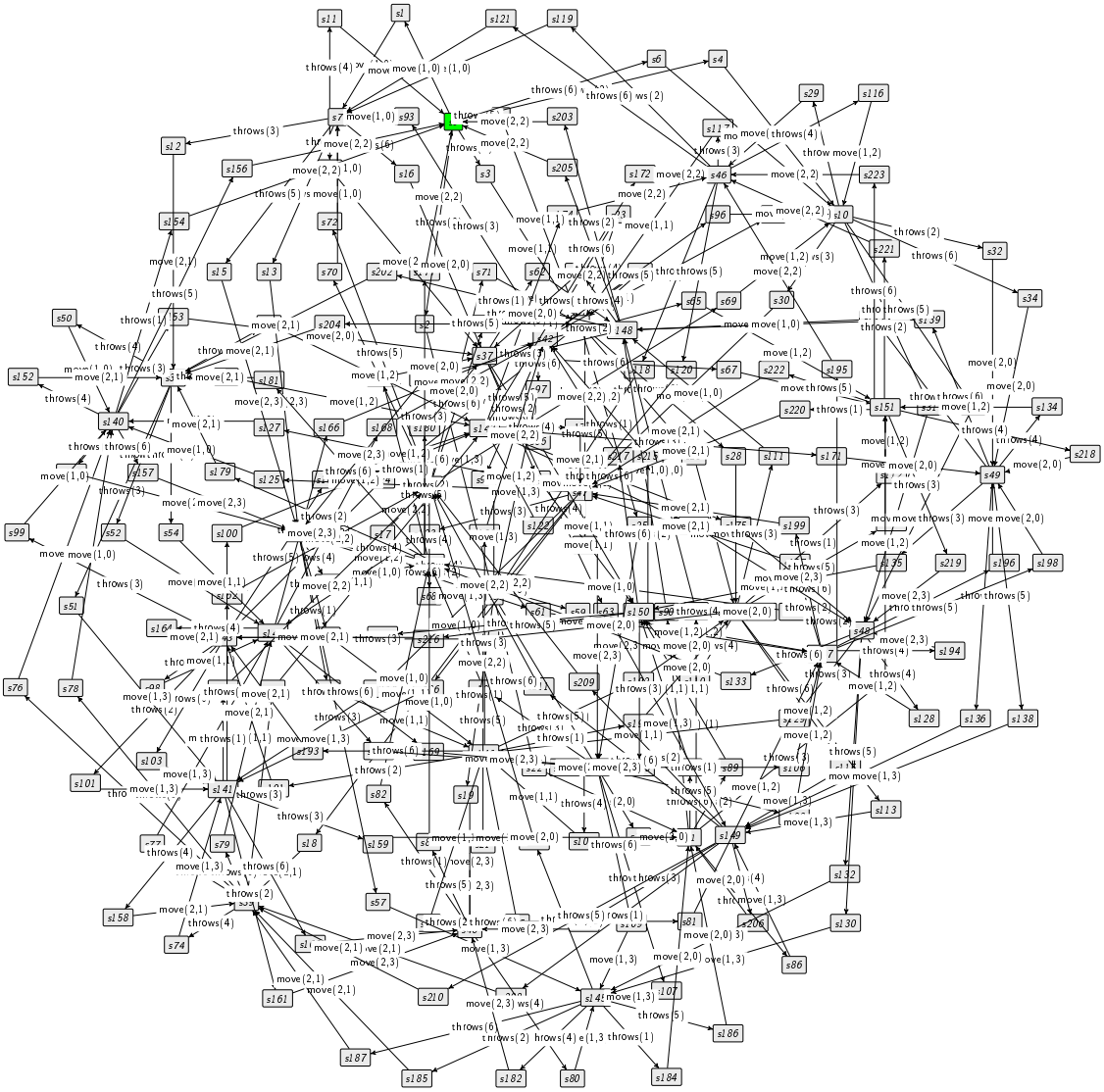


Figure 15: The GTiS of the model in Figure 11

6 Research methods

6.1 GRATiS Design

The specific method used to achieve the design goal is to use GROOVE as a replacement of the STS in ATM. Figure 16 shows this graphically. The Rule Applier component now communicates with the Interface, which fulfills the role of STS Engine. The communication runs through interfaces on both tools, such that other tools are also able to communicate with the GROOVE component. The possible rule transitions are translated by the Interfaces to possible transitions for the Test Manager. The chosen transitions are translated back to rule transitions by the Interfaces.

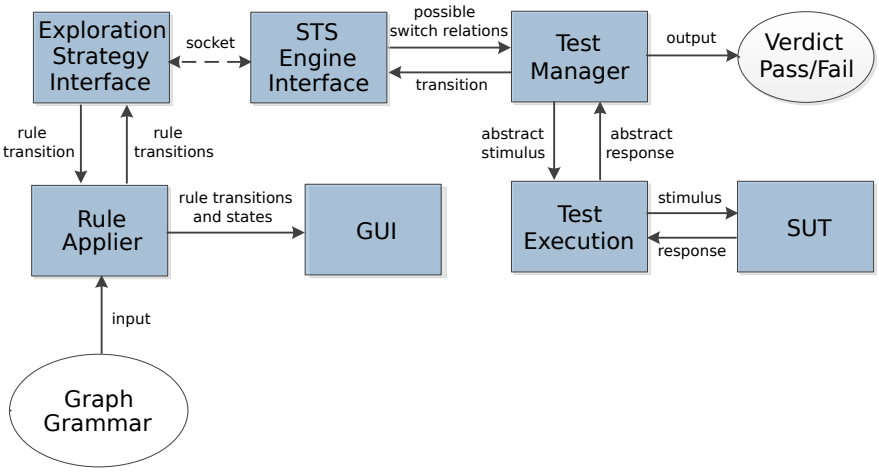


Figure 16: The GRATiS design: replacing the STS with GROOVE

6.1.1 Problems

The boardgame example revealed some problems, listed below, for the GRATiS design. These need to be tackled during the design phase of the project.

- Modelling data types such as integers and strings in GROOVE is problematic; the range of possible values has to be given explicitly and cannot be infinite. For example, it would be impossible to extend the die of the boardgame example such that it can throw any integer; each value of an integer would have to be defined separately.
- In order to efficiently define transitions with a rule system, sometimes one transition is represented by multiple transitions. In the case of the boardgame example, the **Player** node moves to the next **Location** node a number of times equal to the number thrown with the die, as shown in Figure 14. A concrete response from the SUT, where the player moves to any location but the next, would need to be translated to a number of 'move' transitions as the abstract response. Otherwise the Test Manager observes this action as erroneous behavior. This translation introduces coupling between the Test Execution component and the specific model. The modularity of these components is therefore lost.

6.1.2 Coverage

ATM annotates the locations it has seen and the switch relations it has chosen in the STS. These annotations are needed to calculate the coverage statistics. GRATiS must therefore be able to determine the location of a graph state, by abstracting from the concrete data values in the graph state.

Also, as mentioned in section 3.4, the calculation of data coverage for a test run will be implemented in GRATiS.

6.1.3 Initial design step: GRATiS-Min

Before the problems in 6.1.1 are tackled and the design of GRATiS implemented, an initial design that minimally achieves the design goal is implemented. This initial design, named GRATiS-Min performs the following steps:

1. generate the GTiS based on the GTS using GROOVE

2. transform the GTiS to a basic STS
3. send the STS to ATM
4. perform the automatic test generation on the STS

Step 2 requires a proof: Consider the GTiS as an LTS. For each state $q \in Q$ create a location $l \in L$. The start location $l_0 \in L$ is the location created by the state $q_0 \in Q$. Make $\mathcal{V}$ an empty set and $\imath$ an empty mapping. For each label $l \in L$ create a gate $\lambda \in \Lambda$. Derive the parameters on the label l using the GTiS. For each parameter $p \in P$ create an interaction variable $i \in \mathcal{I}$ of the same type as p and add i to $param(\lambda)$. Set $arity(\lambda)$ to the number of parameters on the label. For each transition $t \in T$ create a switch relation $r \in \rightarrow$, where:

- the source and the target locations of the switch relation are the locations created by the source and target states of the transition.
- the gate of the switch relation is the gate created by the label on the transition.
- for each $p \in P$ and $i \in \mathcal{I}$ created by p create an $i = p$ expression. The guard of the switch relation is those expressions joined by the $\wedge$ operator.
- create an empty update mapping for the switch relation.

An example of such a transformation is done using the LTS in Figure 5b. The basic STS resulting from the transformation is in Figure 17.

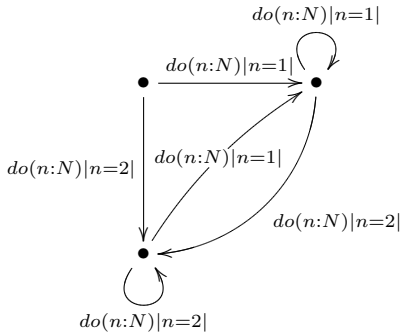


Figure 17: The basic STS resulting from the transformation of the LTS in Figure 5b

This basic STS maintains the behavior of the system. There are a number of advantages to GRATiS-Min:

- It allows automatic test generation on models which GROOVE is able to fully explore. This has been done on some interesting systems, as described in section 5.2.
- It is easy to implement.
- It already implements some features of GRATiS, such as the communication between GROOVE and ATM, which makes GRATiS easier to build.

And also some disadvantages:

- It cannot provide test results on systems with too many states and transitions (for example, due to data values such as integers)
- The Test Manager component of ATM cannot use the data values for its test selection strategy, as all values are spread across multiple switch relations. Also, the calculation of location and switch relation coverage statistics on the basic STS will actually reveal the state and transition coverage.

The last disadvantage can be solved by providing a better transformation of an LTS to an STS. Constructing such a transformation is a sub-goal of the main design goal, as it improves GRATiS-Min. Therefore, this will be part of the research project.

6.2 Validation

There are two steps in the validation:

1. The boardgame example is tested. A SUT is made for this game and ATM is used to test this game. Then, GRATiS is used to test the boardgame. The results should reveal no errors, fail verdicts or other differences in the output of both tools. An intentional error is then made in the SUT and the process is repeated. Still no errors or differences are expected, but both tools should find the error and give a fail verdict.
2. Next, the assessment of the strengths and weaknesses of GRATiS is done by applying both tools to several case studies and comparing the results. The case studies are set out first and then the criteria for the comparison are given.

6.2.1 Case studies

Three case studies are planned. They are all real-life systems Axini has worked on:

- a self-scan register
- a navigation system
- a health-care system

The self-scan register is a machine that automates the purchase of products at a supermarket. A customer can put his products on a conveyer belt and the system automatically calculates the price of the products. Then the customer pays and gets a receipt. The navigation system is a GPS device with a route planner. It allows a user to enter a destination and the system plans the correct route accordingly from the location of the user. The health-care system is a medical device.

A GTS and an STS will be created for each system. GRATiS and ATM are used for the automatic test generation on these models respectively. Both the model and the test process will then be compared as part of the validation. The next two sections give the criteria for this comparison.

6.2.2 Objective criteria

As mentioned in section 1.2, the criteria for the comparison of ATM and GRATiS are split in two parts. The objective criteria that can be compared are:

1. The verdicts: same test cases should give same verdicts. When a verdict is different for the same test case in GRATiS and ATM, this might indicate an error.
2. The number of bugs found: one tool could generate smarter test-cases and find more bugs.
3. The coverage: generating smarter test-cases can also lead to a higher coverage.
4. The test cases generated per second: benchmarks will be done on how fast the tools generate test-cases.
5. The size of the statespace: benchmarks will be done on how much space the tools use.

6.2.3 Subjective criteria

Subjective criteria, such as the maintainability and extendibility of a model, are related to the usability of GTSs versus alternative formalisms, such as STSs. Evaluaiting such criteria requires an extensive experiment with a statistical significant set of human actors. It is out of the scope of this project to perform such an experiment in order to compare two modelling formalisms. In section 1 a motivation is given for the use of GTSs in model-based testing. The qualities of GTSs described there cover the usability of the GTSs.

References

[1] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-JÃűrg Kreowski, Sabine Kuske, Detlef Plump, Andy SchÃijrr, and Gabriele Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 34(1):1 – 54, 1999.

[2] Axel Belinfante. Jtorx: A tool for on-line model-driven test derivation and execution. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 266–270. Springer Berlin / Heidelberg, 2010.

[3] Ed Brinksma and Jan Tretmans. Testing transition systems: An annotated bibliography. In Franck Cassez, Claude Jard, Brigitte Rozoy, and Mark Ryan, editors, *Modeling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 187–195. Springer Berlin / Heidelberg, 2001. 10.1007/3-540-45510-8_9.

[4] Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. Stg: A symbolic test generation tool. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 151–173. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-46002-0_34.

[5] A. H. Ghamarian, M. J. de Mol, A. Rensink, E. Zambon, and M. V. Zimakova. Modelling and analysis using groove. *International journal on software tools for technology transfer*, online pre-publication, March 2011.

[6] Hasan and Ural. Formal methods for test sequence generation. *Computer Communications*, 15(5):311 – 325, 1992.

[7] R. Heckel. Graph transformation in a nutshell. *Electronic Notes in Theoretical Computer Science*, 148(1 SPEC. ISS.):187–198, 2006. cited By (since 1996) 16.

[8] Tim A.C. Willemse Lars Frantzen, Jan Tretmans. Test generation based of symbolic specifcations. Technical report, Nijmegen Institute for Computing and Information Sciences (NIII), 2005.

[9] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090 –1123, aug 1996.

[10] R. Busser M. Blackburn and A. Nauman. Why model-based test automation is diiňĂerent and what you should know to get started. *International Conference on Practical Software Quality and Testing*, 2004.

[11] Joost-Pieter Katoen Manfred Broy, Bengt Jonsson and Alexander Pretschner. *Model-Based Testing of Reactive Systems*. Springer-Verlag Berlin Heidelberg, 2005.

[12] Thomas J. McCabe. Structured testing: A software testing methodology using the cyclomatic complexity metric, 1982. *National Bureau of Standards, Special Publication*, 1982.

[13] Steve McConnell. Software quality at top speed. *Softw. Dev.*, 4:38–42, August 1996.

[14] Lev Nachmanson, Margus Veanes, Wolfram Schulte, Nikolai Tillmann, and Wolfgang Grieskamp. Optimal strategies for testing nondeterministic systems. *SIGSOFT Softw. Eng. Notes*, 29:55–64, July 2004.

[15] R. De Nicola and M.C.B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(1-2):83 – 133, 1984.

[16] Martin Pol. *Testen volgens Tmap (in dutch, Testing according to Tmap)*. Uitgeverij Tutein Nolthenius, 1995.

[17] Vlad Rusu, Lydie du Bousquet, and Thierry Jéron. An approach to symbolic test generation. In Wolfgang Grieskamp, Thomas Santen, and Bill Stoddart, editors, *Integrated Formal Methods*, volume 1945 of *Lecture Notes in Computer Science*, pages 338–357. Springer Berlin / Heidelberg, 2000. 10.1007/3-540-40911-4_20.

[18] G. J. Tretmans and H. Brinksma. Torx: Automated model-based testing. In A. Hartman and K. Dussa-Ziegler, editors, *First European Conference on Model-Driven Software Engineering, Nuremberg, Germany*, pages 31–43, December 2003.

[19] Jan Tretmans. *A Formal Approach to Conformance Testing*. PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

[20] Jan Tretmans. Test generation with inputs, outputs, and quiescence. In Tiziana Margaria and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 127–146. Springer Berlin / Heidelberg, 1996. 10.1007/3-540-61042-1_42.